A Framework For Migrating Applications Under Changing Load Conditions In The Grid*

Sathish S. Vadhiyar and Jack J. Dongarra

Computer Science Department
University of Tennessee, Knoxville
{vss, dongarra}@cs.utk.edu

Abstract. There have been many research efforts that built migration systems which migrate applications under different conditions like load changes on machines, availability of new machines, non-availability of existing machines due to reclaiming by owners, providing fault tolerance etc. Due to the dynamics involved in computational grids, the main motivations for migration of executing applications in grid systems are providing fault tolerance and adapting to changes in machine loads. In this paper we describe our framework that implements scheduling policies for migrating executing applications due to load changes on the resources. Our framework makes migrating decisions based on both system load and application characteristics. We also present some results that validate the usefulness of our migrating system.

1 Introduction

The ability to migrate executing applications onto different sets of resources provides flexibility to resource management systems to efficiently schedule the applications on the system under changing system conditions. Different migration systems [17], [1], [10], [16], [21], [8], [7], [23], [9], [26], [14], [4] have been implemented to migrate different kinds of applications under different conditions. Four issues have to be dealt to build efficient migration systems.

1. *When* - The scheduling and migrating systems have to define the conditions under which migration of executing applications will take place. These conditions can be few key strokes on the executing systems, sudden non-availability of the systems on which the applications are executing, availability of new sets of resources, load imbalance on the systems etc.
2. *Where* - After the decision to migrate, the scheduling system should determine the new sets of resources on which the applications will be migrated. These new sets of resources can be determined based on different sets of criteria.

* This work is supported in part by the National Science Foundation contract GRANT #E81-9975020, SC R3605-29200099, R01-1030-09.

3. *How* - Different migrating systems employ different methods for migrating applications for different kinds of applications. Some migrations can be simple context switches while some migrations can involve complex checkpointing mechanisms.
4. Who - The migration decisions and the migration process can be implemented by the system automatically or can be specified by the user.

Computational grids [12] involve large system dynamics that migration of executing applications on the grid systems assumes more significance. Specifically, the main motivations for migrating applications in grid systems are to provide fault tolerance and adapting to change in loads on the systems. In this paper, we focus on migration of grid applications when the loads on the grid resources change. Many existing migration systems implement policies to migrate applications under changing system loads [17], [10], [16], [23], [26], [14], [4]. Some of these policies are not clearly defined and some other policies are inadequate to the grid systems. The policies when implemented on grid system can involve large scheduling overheads for the grid scheduling and resource management systems.

In this paper, we describe a framework that defines and implements scheduling policies for migrating grid applications in response to system load changes. In our framework, the migration of applications depends on

1. the amount of increase or decrease in loads on the resources,
2. the time of the application execution when load is introduced into the system,
3. the performance benefits that can be obtained for the application due to migration.

Thus, our migrating framework takes into account both the load and application characteristics. The policies are implemented in such a way that the overhead to the grid scheduling system is minimal. The framework has been implemented and tested on top of the GrADS system [2]. Our test results indicate that our migrating system is useful for applications on the grid.

In Section 2, we describe the GrADS system and the life cycle of GrADS applications. In Section 3, we introduce our migration framework by describing the different components for migration. In Section 4, we describe our experiments and provide various results. In Section 5, we present related work in the field of migration. We give concluding remarks in Section 6 and we explain our future plans in Section 7.

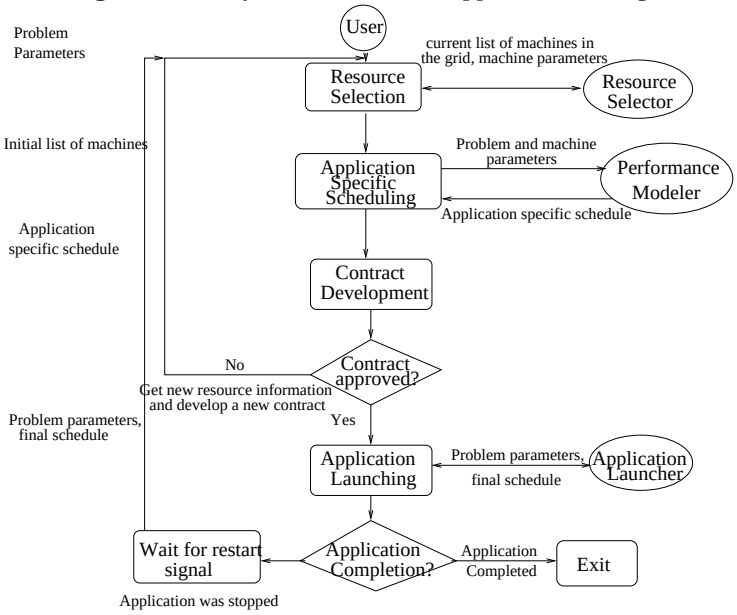
2 The GrADS System

GrADS [2] is an ongoing research involving number of institutions and its goal is to simplify distributed heterogeneous computing in the same way that the World Wide Web simplified information sharing over the Internet. The University of Tennessee investigates issues regarding integration of numerical libraries in the GrADS system. In our previous work [18], we demonstrated the ease with which

numerical libraries like ScaLAPACK can be integrated into the Grid system and the ease with which the libraries can be used over the Grid. We also showed some results to prove the usefulness of a Grid in solving large numerical problems.

In the architecture of GrADS, the user wanting to solve a numerical application over the grid invokes the GrADS application manager. The life cycle of the GrADS application manager is shown in Figure 1.

Figure 1. Life cycle of the GrADS application manager



As a first step, the user invokes the application manager with the problem he wants to solve along with the problem parameters. The application manager invokes a component called Resource Selector. The Resource Selector accesses the Globus Monitoring and Discovery Service(MDS) [11] to get a list of machines in the GrADS testbed that are alive and then contacts the Network Weather Service(NWS) [24] to get system information for the machines. The application manager then invokes a component called Performance Modeler with problem parameters, machines and machine information. The Performance Modeler using an execution model built specifically for the application, determines the final list of machines for application execution. By employing the application specific execution model, GrADS follows the AppLeS [3] approach to scheduling. The problem parameters and the final list of machines are passed as a contract to a component called Contract Developer. The Contract Developer can either approve or reject the contract. If the contract is rejected, the application manager develops a new contract by starting from the resource selection phase again. If

the contract is approved, the application manager passes the problem, its parameters and the final list of machines to Application Launcher. The Application Launcher spawns the job on the given machines using Globus job management mechanism and also spawns a component called Contract Monitor. The Contract Monitor through an Autopilot mechanism [20] monitors the times taken for different parts of applications. The GrADS architecture also has a GrADS Information Repository(GIR) that maintains the different states of the application manager and the states of the numerical application. After spawning the numerical application through the Application Launcher, the application manager waits for the job to complete. The job can either complete or suspend its execution due to an external mechanism. These application states are passed to the application manager through the GIR. If the job has completed, the application manager exits, passing success values to the user. If the application is stopped, the application manager waits for a resume signal and then collects new machine information by starting from the resource selection phase again.

3 The Migration Framework

The ability to migrate applications in the GrADS system is implemented by adding a component called *Rescheduler* to the GrADS architecture The migrating numerical application, *migrator*, the *contract monitor* that monitors the application's progress and the *rescheduler* that decides when to migrate, together form the core of the migrating framework. These components are described in detail in the following subsections.

3.1 The Migrator

Mechanisms have been implemented in the ScaLAPACK LU and QR factorization codes that will enable the ScaLAPACK application to be stopped and restarted on possibly different number of processors. The checkpointing mechanism is application specific in that calls have to be inserted in the applications to checkpoint the data and the application has to be modified with conditional statements to make it work in both the start and restart modes. We use the Internet Backplane Protocol (IBP) [19] for storage of the checkpoint states. IBP depots, where storage can be allocated, are started on the processors of the Grid System.

A component, called Runtime Support System (RSS) is started in the application launching phase even before the application is started. The RSS is specific to the ScaLAPACK application and registers to the GrADS Information Repository(GIR) with the application identifier. When the application is started, it gets the location of the RSS from the GIR and contacts the RSS and performs some initialization. The RSS can receive a STOP signal from any external component. The application, between the iterations, contacts the RSS to check if it has received the STOP signal. When the STOP signal is received, each process of the application, by means of IBP client API, opens up a storage of specific

size in its IBP depot and checkpoints the data to the IBP storage. The handles to the stored data are stored in the RSS. When the application is restarted on a different set of processors, the application contacts the RSS through the GIR, obtains the handles to the IBP data, reads in the checkpointed data, continues from its previous state and proceeds to completion. When the number of processors in the restarted application is different from the number of processors in the original configuration, the RSS allocates new IBP storage and redistributes data from the previous IBP storage to the new storage by means of IBP client API for copying.

3.2 Contract Monitor

Contract Monitor is a component that uses the Autopilot infrastructure to monitor the progress of the applications in GrADS. Autopilot [20] is a real-time adaptive control infrastructure built by the Pablo group at University of Illinois, Urbana-Champagne. An autopilot manager is started before the launch of the numerical application. The numerical application is instrumented with calls to register to the autopilot. The contract monitor retrieves the registration information of the application through the autopilot. The ScaLAPACK applications are also instrumented with calls between each iteration to send the times taken for the iterations to the contract monitor. The contract monitor compares the actual iteration times with the predicted iteration times and calculates the ratio between them. The tolerance limits of the ratio are specified as inputs to the contract monitor.

When a given ratio is greater than the upper tolerance limit, the contract monitor calculates the average of the computed ratios. If the average is greater than the upper tolerance limit, it contacts the rescheduler, requesting for migrating the application. The average of the ratios is used by the contract monitor to contact the rescheduler due to the following reasons:

1. A competing application of short duration on one of the machines may have increased the load on the machine and hence the loss in performance of the application. Contacting the rescheduler for migration on noticing few losses in performance will result in unnecessary migration in this case since the competing application will end soon and the application's performance will be back to normal.
2. The average of the ratios also captures the history of the behavior of the machines on which the application is running. If the application's performance on most of the iterations has been satisfactory, then few losses of performance may be due to sparse occurrences of load changes on the machines.
3. The average of the ratios also takes into account the percentage completed time of application's execution.
4. Contacting the rescheduler for migration only when the average of ratios is greater than the upper tolerance limit significantly reduces the overhead of migrating decisions.

When the rescheduler refuses to migrate the application, the contract monitor adjusts its tolerance limits to new values. Similarly when a given ratio is less than the lower tolerance limit, the contract monitor calculates the average of the ratios and adjusts the tolerance limits if the average is less than the lower tolerance limit. The dynamic adjusting of tolerance limits serves three purposes:

1. It reduces the overhead involved in contract monitor when the ratios between actual and predicted times are not the original expected ratios.
2. It reduces the amount of communication between the contract monitor and the rescheduler.
3. It hides the deficiencies in the application-specific execution time model.

3.3 Rescheduler

Rescheduler is the component that evaluates the performance benefits that can be obtained due to the migration of an application and initiates the migration of the application. The rescheduler is a daemon that operates in 2 modes: *migration on request* and *opportunistic migration*. When the contract monitor detects intolerable performance loss for an application, it contacts the rescheduler requesting it to migrate the application. This is called migration on request. In other cases when no contract monitor has contacted the rescheduler for migration, the rescheduler periodically queries the GrADS Information Repository(GIR) for recently completed applications. If a GrADS application was recently completed, the rescheduler determines if performance benefits can be obtained for an executing application by migrating it to use the resources that were freed by the completed application. This is called opportunistic rescheduling.

In both cases, the rescheduler first contacts the Network Weather Service (NWS) to get the updated information for the machines in the grid. It then contacts the application-specific performance modeler to evolve a new schedule for the application. Based on the total percentage completion time for the application and the total predicted execution time for the application with the new schedule, the rescheduler calculates the remaining execution time, *ret_new*, of the application if it were to execute on the machines in the new schedule. The rescheduler also calculates *ret_current*, the remaining execution time of the numerical application if it were to to continue executing on the original set of machines. *ret_current* is calculated based on the progress of the application and the total predicted time of execution for the application on the original set of machines. The rescheduler then calculates the rescheduling gain as

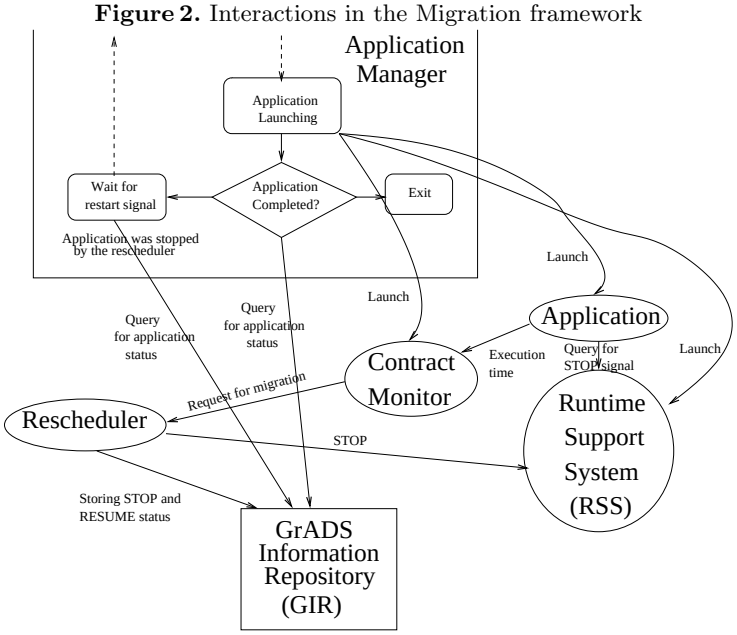
$$rescheduling\_gain = \frac{(ret\_current - (ret\_new + 240))}{ret\_current}$$

where 240 is the worst case time in seconds needed to reschedule the application. This involves the time for the application manager to pass the phases of resource selection, application-specific rescheduling, contract development and application launching and the time for the redistribution of data if the application is restarted on a different number of processors. If the rescheduling

gain is greater than 30%, the rescheduler sends STOP signal to the application, and stores the stop status in GIR. The application manager then waits for the RESUME signal. The rescheduler stores the RESUME value in the GIR thus prompting the application manager to evolve a new schedule and restart the application on the new schedule. If the rescheduling gain is less than 30% and if the rescheduler is operating in the *migration on request* mode, the rescheduler contacts the contract monitor prompting the contract monitor to adjust its tolerance limits.

The interactions between the different components involved in the migration framework is illustrated in Figure 2.

Figure 2. Interactions in the Migration framework



4 Experiments and Results

The GrADS experimental testbed consists of about 40 machines that reside in institutions across the country including University of Tennessee, University of Illinois, University of California at San Diego, Rice University etc. For the sake of clarity, our experimental testbed consists of two clusters in University of Tennessee, a cluster called *torc* consisting of 8 machines and another cluster called *msc* consisting of 8 machines. The two clusters are connected to each other by 100 Mb Ethernet links. Both the clusters consist of Pentium III machines running Linux and the machines within a cluster are connected to each other by

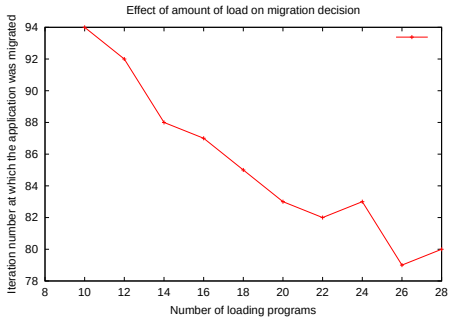
100 Mb switched Ethernet. Machines in *torc* have clock speed of 550 MHz while machines in *msc* have clock speed of 933 MHz.

In the following experiments, we demonstrate both opportunistic migration and migration on request. The loading program we used to demonstrate migration on request is a simple C code that consist of a single looping statement that loops forever. This program was compiled without any optimization in order to achieve the loading effect. For all the experiments, ScaLAPACK QR factorization was used.

4.1 Migration on Request

In all the experiments in this section, 8 *torcs* and 4 *mscs* were used. Since, *mscs* are faster than *torcs*, all the 4 *mscs* were used for the initial schedule. The loading programs were introduced in the 4 *mscs*.

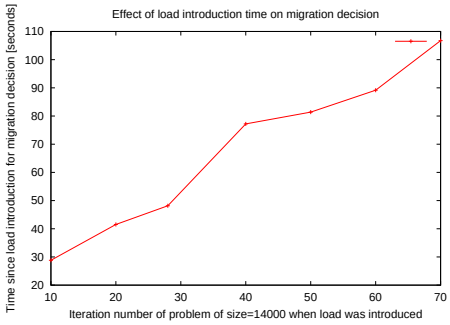
Figure 3. Effect of amount of load on migration decision



In the first set of experiments, we demonstrate the effect of amount of load on the migration decision. In these experiments, we use the same problem size, matrix size of 14000, and introduce the load at the same point in application execution, at iteration number 70. The total number of iterations in the application is 375. In Figure 3, the x-axis represents the number of loading programs used in the system and the y-axis represents the iteration in the application execution when the application was migrated. We note that more the amount of load introduced in the system, earlier the application is migrated.

In the second set of experiments, we illustrate that the migrating decisions can be affected by the time in the application execution when the load was introduced into the system. In these experiments, we use the same problem size, matrix size of 14000, and the same amount of load, 10 instances of the loading program. In Figure 4, the x-axis denotes the iteration number in the application when the load was introduced and the y-axis denotes the difference in time in seconds between when the load was introduced and when the rescheduler decided to migrate the application.

Figure 4. Effect of load introduction time on migration



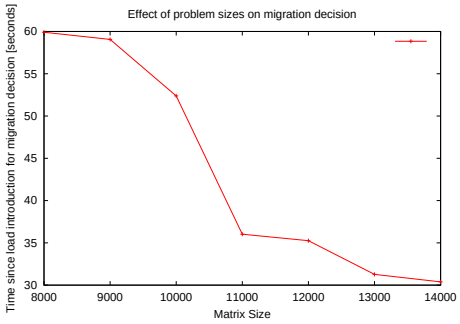
From Figure 4, we observe that the scheduling system takes more time for deciding to migrate as the time for load introduction into the system increases. This is due to the effect of history of application's behavior on migration. When the load is introduced at iteration 70, the first 70 iterations have been executed according to expected behavior and a sustained loading effect of lengthy duration is needed to convince the scheduling system to migrate the application. For iterations greater than 70, the rescheduler refused to migrate the application since very little rescheduling gain can be obtained. For these cases, the contract monitor dynamically adjusted its tolerance limits.

For the third set of experiments, we demonstrate that the same amount of load can have different effects on the migration decision when different problems of different sizes were used. For these experiments, we introduce the same amount of load, 10 loading programs, into the systems 3 minutes after the start of the application execution. In Figure 5, the time taken for deciding to migrate the application since the load introduction is plotted against the different matrix sizes of the problem. At the time of introduction of the load, the smaller applications would have progressed more than the larger applications. Hence migration decision is immediate in the case of larger applications. Also, for applications of matrix sizes less than 8000, the rescheduler decided not to migrate since significant performance benefits will not be obtained due to migration.

4.2 Opportunistic Migration

In this set of experiments, we illustrate opportunistic migration in which the rescheduler tries to migrate an executing application when some other application completes. An application, app_1, was introduced into the system such that it consumed most of the memory of 8 *msc* machines. During the execution of app_1, an app_2, that intended to use 11 machines, 3 *torcs* and 8 *mscs* was introduced into the system. Since the 8 *msc* machines were occupied by app_1, app_2 was able to utilize only the 3 *torc* machines. When app_1 completed, the 8 *msc* machines were freed and app_2 was able to utilize the extra resources to reduce its

Figure 5. Effect of the same load amount on different problem sizes



remaining execution time. The rescheduler evaluated the performance benefits of allowing app_2 to utilize the extra 8 processors.

ScaLAPACK problems of sizes 20000 and 21000, depending on the available memory on *mscs* when the experiments were run, were used for app_1. ScaLAPACK problem of size 11000 was used for app_2.

We define

1. Total execution time of app_2 on 3 *torcs* without rescheduling, $exec_{without\_re}$
2. Total execution time of app_2 with rescheduling, $exec_{with\_re}$
3. Percentage rescheduling gain for app_2, $percentage_gain$

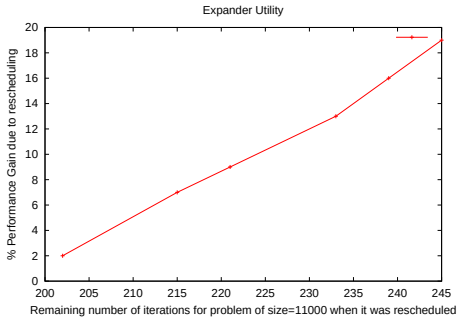
$$percentage\_gain = \frac{exec_{without\_re} - exec_{with\_re}}{exec_{without\_re}}$$

app_2 was introduced at various points of time after the starting of app_1. Hence additional resources will be available for app_2 at various points of time into its execution. The total number of iterations needed by the ScaLAPACK problem of size 11000 was 275. Figure 6 illustrates the utility of rescheduling as a function of the remaining number of iterations left for app_2 when app_2 was rescheduled. We observe that the percentage rescheduling gain for app_2 increases when the remaining execution time left for app_2 at the time of rescheduling increases.

5 Related Work

Different systems have been implemented to migrate executing applications onto different sets of resources. The work by Mirchandaney et. al. [17] deals with migration of executing applications to efficiently use under-utilized resources. The Dome system [1] performs data redistribution for load balancing and migrates executing applications to provide fault resilience. Sprite operating system [10] and Condor [16] perform checkpointing and migration of executing applications when the owner of a workstation wants to reclaim his system. Thus the migration policy is intended to reduce the obtrusiveness to the workstation owner and not to increase the performance of individual applications as implemented

Figure 6. Rescheduling gain for app_2



in our migration framework. Khaled Al-Saqabi et. al [21] discusses migration of applications in the context of gang scheduling. MPVM/MIST [8], [7] projects and the work by Zhang et. al. [25] have built migration systems that uses the concept of gang scheduling to utilize system resources. MIST also deals with migration under increasing loads but the scheduling policy has not been defined clearly. The Dynamite system [23] based on Dynamic PVM [9] migrates applications when the loads of certain machines gets under-utilized or over-utilized as defined by application-specified thresholds. Although this method takes into account application-specific characteristics it does not necessarily evaluate the remaining execution time of the application and the resulting performance benefits due to migration. The migration system in LSF[26] migrates jobs under load changes if a migration threshold is defined for the hosts. Thus jobs are migrated to maintain load balance rather than to improve performance. MARS [14] migrates applications taking into account both the system loads and application characteristics. But the migration decisions are made only at different phases of the applications. In our migration framework, the applications are continuously monitored and migration decisions are made whenever the applications are not making sufficient progress. The HMF system [4] uses a graph model to define migration policies. The efficiency of this model in grid systems is still to be proven. Of the grid computing systems [13], [15], [16], [6], [5], [22], only Condor [16] seems to migrate applications under workload changes. But as mentioned above, the goal in Condor is to utilize idle resources rather than to improve performance.

6 Conclusions

Many existing migrating systems that migrate applications under loading conditions implement simple policies that cannot be applied to grid systems. We have implemented a migration framework that takes into account both the system load and application characteristics. The migrating decisions are made without adding too much overhead to the scheduling system in the grid. Migration

decisions are based on factors like the amount of load, the time of the application when the load is introduced and the size of the applications. We have also implemented a framework that migrates executing applications to make use of additional free resources. Experiments were conducted and results were presented to demonstrate the capabilities of the migration framework.

7 Future Work

Work is in progress to develop interfaces for the application library writer to to easily integrate his application with migration capability into the GrADS system. The present rescheduler uses the worst case rescheduling time to make its scheduling decisions. Our plan is to make the rescheduler learn the rescheduling cost based on previous runs. We are also investigating providing support for fault tolerance in the GrADS framework.

References

1. J.N.C. Arabe, A.B.B. Lowekamp, E. Seligman, M. Starkey, and P. Stephan. Dome: Parallel Programming in a Heterogeneous Multi-User Environment. *Supercomputing*, 1995.
2. F. Berman, A. Chien, K. Cooper, J. Dongarra, I. Foster, D. Gannon, L. Johnsson, K. Kennedy, C. Kesselman, J. Mellor-Crummey, D. Reed, L. Torczon, and R. Wolski. The GrADS Project: Software Support for High-Level Grid Application Development. *International Journal of High Performance Applications and Supercomputing*, 15(4):327–344, Winter 2001.
3. F. Berman and R. Wolski. The AppLeS Project: A Status Report. *Proceedings of the 8th NEC Research Symposium*, May 1997.
4. M. Bishop, M. Valence, and L. F. Wisniewski. Process Migration for Heterogeneous Distributed Systems. Technical Report TR95-264, 1995.
5. R. Buyya, D. Abramson, and J. Giddy. Nimrod-G Resource Broker for Service-Oriented Grid Computing. *IEEE Distributed Systems Online*, 2(7), November 2001.
6. H. Casanova and J. Dongarra. NetSolve: A Network Server for Solving Computational Science Problems. *The International Journal of Supercomputer Applications and High Performance Computing*, 11(3):212–223, Fall 1997.
7. J. Casas, D. Clark, P. Galbiati, R. Konuru, S. Otto, R. Prouty, and J. Walpole. Mist: Pvm with Transparent Migration and Checkpointing, 1995.
8. J. Casas, D. Clark, R. Konuru, S. Otto, R. Prouty, and J. Walpole. MPVM: A Migration Transparent Version of PVM. Technical Report CSE-95-002, 1, 1995.
9. L. Dikken, F. van der Linden, J. J. J. Vesseur, and P. M. A. Sloot. DynamicPVM: Dynamic Load Balancing on Parallel Systems. In Wolfgang Gentzsch and Uwe Harms, editors, *Lecture notes in computer science 797, High Performance Computing and Networking*, volume Proceedings Volume II, Networking and Tools, pages 273–277, Munich, Germany, April 1994. Springer Verlag.
10. F. Douglis and J. K. Ousterhout. Transparent Process Migration: Design Alternatives and the Sprite Implementation. *Software - Practice and Experience*, 21(8):757–785, 1991.

11. S. Fitzgerald, I. Foster, C. Kesselman, G. von Laszewski, W. Smith, and S. Tuecke. A Directory Service for Configuring High-Performance Distributed Computations. volume Proc. 6th IEEE Symp. on High-Performance Distributed Computing, pages 365–375, 1997.
12. I. Foster and C. Kesselman eds. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, ISBN 1-55860-475-8, 1999.
13. I. Foster and C. Kesselman. Globus: A Metacomputing Infrastructure Toolkit. *Intl J. Supercomputer Applications*, 11(2):115–128, 1997.
14. J. Gehring and A. Reinefeld. MARS - A Framework for Minimizing the Job Execution Time in a Metacomputing Environment. *Future Generation Computer Systems*, 12(1):87–99, 1996.
15. A. Grimshaw, W. Wulf, J. French, A. Weaver, and Jr. P. Reynolds. Legion: The Next Logical Step Toward a Nationwide Virtual Computer. Technical Report CS-94-21, Department of Computer Science, University of Virginia, 1994.
16. M. Litzkow, M. Livney, and M. Mutka. Condor - a Hunter for Idle Workstations. *Proc. 8th Intl. Conf. on Distributed Computing Systems*, pages 104–111, 1988.
17. R. Mirchandaney, D. Towsley, and J. A. Stankovic. Adaptive Load Sharing in Heterogeneous Distributed Systems. *Journal of Parallel and Distributed Computing*, 9:331–346, 1990.
18. A. Petitet, S. Blackford, J. Dongarra, B. Ellis, G. Fagg, K. Roche, and S. Vadhiyar. Numerical Libraries and the Grid: The Grads Experiments with Scalapack. *Journal of High Performance Applications and Supercomputing*, 15(4):359–374, Winter 2001.
19. J. S. Plank, M. Beck, W. R. Elwasif, T. Moore, M. Swany, and R. Wolski. The Internet Backplane Protocol: Storage in the Network. *NetStore99: The Network Storage Symposium*, 1999.
20. R.L. Ribler, J.S. Vetter, H. Simitci, and D.A. Reed. Autopilot: Adaptive Control of Distributed Applications. *Proceedings of the 7th IEEE Symposium on High-Performance Distributed Computing*, July 1998.
21. K.A. Saqabi, S.W. Otto, and J. Walpole. Gang Scheduling in Heterogeneous Distributed Systems. Technical report, OGI, 1994.
22. S. Sekiguchi, M. Sato, H. Nakada, and U. Nagashima. Ninf: Network based Information Library for Globally High Performance Computing. *Parallel Object-Oriented Methods and Applications (POOMA)*, February 1996.
23. G.D. van Albada, J. Clinckemaillie, A.H.L. Emmen, J. Gehring, O. Heinz, F. van der Linden, B.J. Overeinder, A. Reinefeld, and P.M.A. Sloot. Dynamite - Blasting Obstacles to Parallel Cluster Computing, April 1995.
24. R. Wolski, N. Spring, and J. Hayes. The Network Weather Service: A Distributed Resource Performance Forecasting Service for Metacomputing. *Journal of Future Generation Computing Systems*, 15(5-6):757–768, October 1999.
25. Y. Zhang, H. Franke, J. Moreira, and A. Sivasubramaniam. The impact of migration on parallel job scheduling for distributed systems. In *Lecture Notes in Computer Science 1900*, volume 6th International Euro-Par Conference, pages 242–251, Aug/Sep 2000.
26. S. Zhou, X. Zheng, J. Wang, and P. Delisle. Utopia: a Load Sharing Facility for Large, Heterogeneous Distributed Computer Systems. *Software – Practice and Experience*, 23(12):1305–1336, December 1993.